Algoritmos y Estructura de Datos: Examen 1 (Solución)

Grados Ing. Inf. y Mat. Inf. Abril 2011

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería de Software

Apellidos: .....

Nombre: .....

DNI / NIE: ..... **Núm. matrícula:** .....

Normas

- Este examen consta de **4 preguntas** en **7 páginas**.
- La puntuación total del examen es de **10 puntos**.
- La duración total del examen es de **60 minutos**.
- El examen debe contestarse **en las hojas que se proporcionan**.
- Deben rellenarse los campos obligatorios **apellidos, nombre, y DNI/NIE**.
- Las calificaciones provisionales de este examen se publicarán en el Aula Virtual el **25 de abril de 2012** junto con las soluciones. La fecha de la revisión de este examen es el **27 de abril de 2012**. La hora y lugar de dicha revisión se anunciará en el Aula Virtual.
- En las preguntas con varias opciones, **sólo hay una respuesta válida por pregunta**. En este caso toda pregunta en que se marque más de una respuesta se considerará incorrectamente contestada y toda pregunta incorrectamente contestada restará del examen una cantidad de puntos igual a la puntuación de la pregunta dividido por el número de alternativas ofrecidas en la misma menos uno. Es decir, una respuesta incorrecta en una pregunta de un punto con cuatro alternativas resta $\frac{1}{3}$ de punto.

(1 punto) 1. Se tiene un videojuego de carreras de coches en el que pueden correr hasta un máximo de 20 coches. Se sabe que calcular la siguiente posición de *un* coche es una operación que tiene complejidad constante. El videojuego dispone de un algoritmo que toma un vector con las posiciones de *todos* los coches y calcula la siguiente posición de cada coche, recorriendo el vector desde el primer coche hasta el último. **Se pide:** Indicar la complejidad en caso peor de dicho algoritmo, razonando la respuesta.

> La complejidad de recorrer completamente un vector es $O(n)$ con n el tamaño del vector. Sin embargo, $n \leq 20$, es decir, el tamaño del vector está acotado. En el caso peor la complejidad será $O(20)$ que es lo mismo que $O(1)$, es decir, complejidad constante.

(2 puntos) 2. Se ha implementado un método `findLast` que toma dos parámetros: un objeto de una clase que implementa el interfaz `PositionList` y un objeto elemento de tipo genérico `E`. El método debe devolver el nodo donde se encuentra la <u>última</u> ocurrencia del elemento en la lista. Cuando el elemento no está en la lista el método debe devolver `null`. A continuación se muestra la implementación, que es incorrecta:

```
public Position<E> findLast(PositionList<E> list, E e) {
  Position<E> p;
  for ( p = list.last(); !p.element().equals(e); p = list.prev(p)) ;
  return p.element().equals(e) ? p : null;
}
```

Se sabe que en tiempo de ejecución el objeto `list` siempre será de clase `NodePositionList<E>`. **Se pide:** Indicar todas las invocaciones de métodos en el código que pueden ser causa de excepciones definidas en el paquete `net.datastructures`, razonando brevemente la respuesta. El interfaz `PositionList<E>` está disponible en el Apéndice A.1.

1. La invocación `list.last()` lanzara una excepción `EmptyListException` si la lista parámetro es vacía.
2. La invocación `list.prev(p)` lanzará una excepción `BoundaryViolationException` si `p` es el primer nodo de la lista. Esto último ocurre cuando el elemento buscado no está en la lista.

Otras excepciones posibles tales como `NullPointerException` no son parte del paquete `net.datastructures`.

(3 puntos) 3. Se ha redefinido la clase `BitVectSet<E>`, que implementa conjuntos acotados de elementos enumerados, añadiéndole un método `iterator` que devuelve un iterador sobre los elementos del conjunto:

```
import java.util.Iterator;
public class BitVectSet<E extends Enum<E>> implements Set<E>, Iterable<E> {
  protected boolean bv[];
  protected int size;

  ...
  public Iterator<E> iterator();
}
```

Se pide: Implementar en Java el método `public String[] toArray(BitVectSet<String> s)` que debe devolver un vector («array») con los elementos (cadenas de caracteres) almacenados en el conjunto `s` en el orden en el que son recorridos por su iterador. El interfaz `Set<E>` está disponible en el Apéndice A.3.

Solución que usa «for-each» (la clase `BitVectSet<E>` implementa el interfaz `Iterable<E>`):

```
public String[] toArray(BitVectSet<String> s) {
  if (s == null || s.isEmpty()) return null;
  else {
    String[] r = new String[s.size()];
    i = 0;
    for (String e : s) r[i++] = e;
    return r;
  }
}
```

Solución que usa un iterador explícito:

```
public String[] toArray(BitVectSet<String> s) {
  if (s == null || s.isEmpty()) return null;
  else {
    String[] r = new String[s.size()];
    for (int i = 0, Iterator<E> it = s.iterator(); it.hasNext(); i++)
      r[i] = it.next();
    return r;
  }
}
```

(4 puntos) 4. Se decide añadir a la clase `NodePositionList<E>` el método `append`:

```
package net.datastructures;
import java.util.Iterator;
```

```
public class NodePositionList<E> implements PositionList<E> {
  protected int numElts;              // Number of elements in the list
  protected DNode<E> header, trailer; // Special sentinels
  ...
  void append(NodePositionList<E> list) { /** COMPLETAR **/ }
}
```

Se pide: Implementar en Java el método `append` que será similar al implementado en clase de laboratorio con la salvedad de que tras concatenar los nodos `append` debe dejar la lista parámetro `list` vacía. Recordamos que `append` debe concatenar los nodos de `list` a los nodos de la lista sobre la que se invoca el método. La concatenación debe hacerse con complejidad constante, sin copiar los elementos de `list`.

Por ejemplo, se tienen las listas `l1` y `l2` y se invoca `l1.append(l2)`. Si `l2` es vacía entonces el método no tiene efecto, quedando `l1` inalterada. Si `l1` es vacía entonces `l1` debe «quedarse» con los nodos de `l2` y esta última debe quedar vacía. Si las dos listas contienen elementos, el siguiente nodo al último nodo de `l1` debe ahora ser el primer nodo de `l2`, es decir, los nodos de `l2` se concatenan a los de `l1`. La lista `l2` debe quedar vacía.

El interfaz `PositionList<E>` está disponible en el Apéndice A.1 y la clase `DNode<E>` en el Apéndice A.2.

```
void append(NodePositionList<E> list) {
  if (!list.isEmpty()) {
    if (isEmpty()) {
      /* swap headers and trailers */
      DNode<E> tmpHeader, tmpTrailer;
      tmpHeader    = header;
      tmpTrailer   = trailer;
      header       = list.header;
      trailer      = list.trailer;
      list.header  = tmpHeader;
      list.trailer = tmpTrailer;
    } else {
      /* both lists not empty */
      trailer.getPrev().setNext(list.header.getNext());
      trailer.getPrev().getNext().setPrev(trailer.getPrev());
      DNode<E> oldThisTrailer = trailer;  /* saved for reuse */
      trailer = list.trailer;
      /* now empty the argument list */
      list.trailer = oldThisTrailer;
      list.trailer.setPrev(list.header);
      list.header.setNext(list.trailer);
    }
    numElts += list.numElts;
    list.numElts = 0;
  }
}
```

A. Código de apoyo

A.1. Interfaz `PositionList<E>`

```
package net.datastructures;
import java.util.Iterator;
/**
 * An interface for positional lists.
 * @author Roberto Tamassia, Michael Goodrich
 */
public interface PositionList<E> extends Iterable<E> {
  /** Returns the number of elements in this list. */
  public int size();

  /** Returns whether the list is empty. */
  public boolean isEmpty();

  /** Returns the first node in the list. */
  public Position<E> first();

  /** Returns the last node in the list. */
  public Position<E> last();

  /** Returns the node after a given node in the list. */
  public Position<E> next(Position<E> p)
    throws InvalidPositionException, BoundaryViolationException;

  /** Returns the node before a given node in the list. */
  public Position<E> prev(Position<E> p)
    throws InvalidPositionException, BoundaryViolationException;

  /** Inserts an element at the front of the list, returning new position. */
  public void addFirst(E e);

  /** Inserts and element at the back of the list, returning new position. */
  public void addLast(E e);

  /** Inserts an element after the given node in the list. */
  public void addAfter(Position<E> p, E e)
    throws InvalidPositionException;

  /** Inserts an element before the given node in the list. */
  public void addBefore(Position<E> p, E e)
    throws InvalidPositionException;

  /** Removes a node from the list, returning the element stored there. */
  public E remove(Position<E> p) throws InvalidPositionException;

  /** Replaces the element stored at the given node, returning old element. */
  public E set(Position<E> p, E e) throws InvalidPositionException;

  /** Returns an iterable collection of all the nodes in the list. */
  public Iterable<Position<E>> positions();

  /** Returns an iterator of all the elements in the list. */
  public Iterator<E> iterator();
}
```

A.2. Clase `DNode<E>`

```
package net.datastructures;
/**
 * A simple node class for a doubly-linked list.  Each DNode has a
 * reference to a stored element, a previous node, and a next node.
 *
 * @author Roberto Tamassia
 */

public class DNode<E> implements Position<E> {
  private DNode<E> prev, next;  // References to the nodes before and after
  private E element;    // Element stored in this position
  /** Constructor */
  public DNode(DNode<E> newPrev, DNode<E> newNext, E elem) {
    prev = newPrev;
    next = newNext;
    element = elem;
  }
  // Method from interface Position
  public E element() throws InvalidPositionException {
    if ((prev == null) && (next == null))
      throw new InvalidPositionException("Position␣is␣not␣in␣a␣list!");
    return element;
  }
  // Accessor methods
  public DNode<E> getNext() { return next; }
  public DNode<E> getPrev() { return prev; }
  // Update methods
  public void setNext(DNode<E> newNext) { next = newNext; }
  public void setPrev(DNode<E> newPrev) { prev = newPrev; }
  public void setElement(E newElement) { element = newElement; }
}
```

A.3. Interfaz `Set<E>`

```
package setLibrary;

/** Unbounded set interface */
public interface Set<E> {

  /** An 'equals' method for sets assumed on implementations **/

  /** Cardinality **/
  public int size();

  /** Whether the set is empty **/
  public boolean isEmpty();

  /** Empties the set */
  public void reset();

  /** Membership test **/
  public boolean member(E e);

  /** Inserts the element if not already in the set */
  public void add(E e);

  /** Removes the element if already in the set */
  public void remove(E e);

  /** Union with argument set */
  public void union(Set<E> s) throws InvalidSetException;

  /** Intersection with argument set */
  public void intersect(Set<E> s) throws InvalidSetException;

  /** Difference with argument set */
  public void difference(Set<E> s) throws InvalidSetException;
}
```